

Mostly Functional C#

a.k.a. Taming Side Effects (TSE)



Joe Duffy, Midori
August, 2009



Background, Aka Why?

- My (recent) history:
 - “No more free lunch” in 2004, on the CLR Team.
 - Software transactional memory in 2005.
 - Began Parallel LINQ (PLINQ) also in 2005.
 - Evolved into Parallel Extensions to .NET Framework (4.0) in 2007, served as lead / architect.
 - Shipping in Visual Studio 2010.
- And in the meantime, realized we need more fundamental changes to enable *safe* parallelism.
 - Began with “purely functional”.
 - And evolved to “mostly functional” over past 1 ½ years.

Or, in the words of Simon Peyton Jones

- “[A] big programming-language theme over the next ten years will be mechanisms to restrict or control unrestricted side effects. ... One way or another[,] we need to get a handle on those effects.”

--- *Caging the Effects Monster: the next big challenge*,
Simon Peyton Jones, QCon 2008

Talk Overview

- **What is Midori?**
- What is Mostly Functional C#?
 - Object Isolation
 - Other Features
- The Road Ahead: Reality Check
- Q & A

What is Midori?

- Systems incubation under Eric Rudder
- 75 people, almost entirely developers and architects
- New operating system (spawn of Singularity)
- New application programming model
- Large compiler effort
- Legacy free (mostly)

Key Midori Assumptions

- Cloud is integral
 - User-data in the cloud (partially)
 - Services in the cloud
- Clients are not simple
 - Multi-core and many-core
 - Rich functionality with local data
 - Multiple clients per person
- Clients are mobile
 - Occasionally-connected networks
 - High and variable latencies (e.g., network & memory)
 - Power management is essential
- Security, safety, and reliability are crucial

Key Midori Goals and Principles

- Refresh the software stack for modern issues
 - Existing OSES and app models are ~30 years old
- Reliability
 - Memory, type, **concurrency safety** guaranteed
 - “Correct by construction”
 - Responsiveness: “asynchronous everywhere”
- Security
 - Capability-based: no ACLs
 - “Secure by construction”
- Support using hundreds of cores well


Talk Overview

- What is Midori?
- **What is Mostly Functional C#?**
 - Object Isolation
 - Other Features
- The Road Ahead: Reality Check
- Q & A

Design Approach

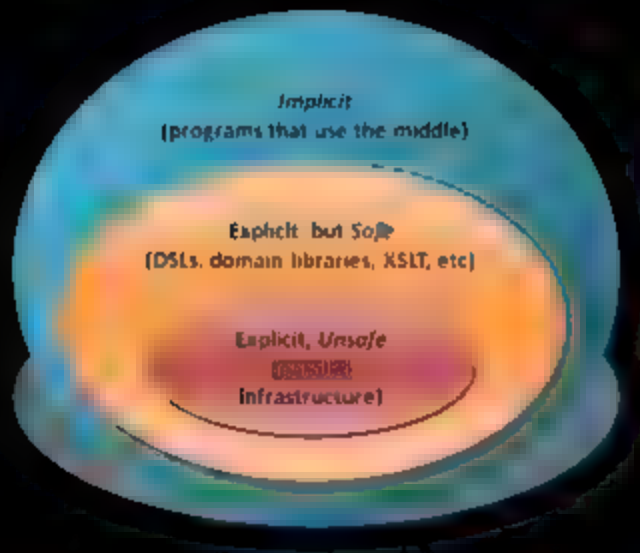
- A prototype extension of the C# language.
- Make functional programming in C# *the default*:
 - C# = { imperative + dynamic + functional }
 - Methods do not mutate existing state.
 - Immutable types are first class.
- Allow imperative programming where familiar, convenient, and/or more efficient:
 - This is still C#! Iteration, objects, etc.
 - State *can* be mutated, but only where declared.
- Why?
 - Deterministic, safe parallelism.
 - Robustness, reliability, and contracts.

Inspiration

- Haskell. 
 - They *can* do implicit parallelism.
 - The safety problem is solved:
 - We can only ever hope to do as good as them, but no better.
 - Parallelism is constrained by *data dependence*.
 - Places emphasis on feedback directed optimizations.
- Launchbury, Jones, Wadler:
 - “Lazy functional state threads.”
 - “Imperative functional programming.”

```
swap :: MutVar s a -> MutVar s a -> ST s ()
swap v w = readVar v    'thenST' \a ->
           readVar w    'thenST' \b ->
           writeVar v b 'thenST_'
           writeVar w a
```

A Landscape of Concurrency



Cut to the Chase

- **Mode #1: Auto-parallelize, when safe:**

```
- var q = from x in xs.AutoParallel()  
  where (x & 1) == 0  
  select x * 0.03f;
```

```
- int c = 0.03f;  
  var q = from x in xs.AutoParallel()  
    where (x & 1) == 0  
    select x * (c += 0.05f);
```

- **Mode #2: Compiler error, when unsafe:**

```
- var q = from x in xs.AsParallel() -  
  where (x & 1) == 0  
  select x * 0.03f;
```

```
- int c = 0.03f;  
  var q = from x in xs.AsParallel()  
    where (x & 1) == 0  
    select x * (c += 0.03f);
```

Feature List (Big-to-Small)

- Object isolation model
- Immutable types
- Non-nullability
- Linear types
- “where $T : S \{ \dots \}$ ” specialization / matching
- Async “future” lifting
- Tuples

A Word of Caution

- *Everything* in MfC# is an experiment.
 - No commitment to add it to C#, in any way.
 - In the process of adopting it in Midori.
 - See how it goes, and let DevDiv know.
- <http://toolbox/> release is coming soon...
 - In the next couple weeks.
 - Complete with a "Programming Guide" overview.
 - And prototype of "safe data parallel" APIs.
 - Join 'tse' alias for updates / announcements.

Talk Overview

- What is Midori?
- What is Mostly Functional C#?
 - Object Isolation
 - Other Features
- The Road Ahead: Reality Check
- Q & A

Key Idea: Object Isolation

- No mutable statics.
- Permission tagged references:
 - **writable** Foo wFoo: all state in the graph can be read/written.
 - **readable** Foo rFoo: all state in the graph can be read.
 - **immutable** Foo iFoo: only readonly fields in the graph can be read.
- Implicit coercion: writable > readable > immutable.
- Default if unstated:
 - Usually readable (functional by default).
 - Sometimes immutable (for references of immutable types).
- Appears in any position (see next slide).

Where Can Permissions Appear?

- Fields:

- `class C { public writable T m_f; }`

- Local variables:

- `public isolated void M() { writable T obj = ... }`

- Formal parameters:

- `public isolated void M(writable T obj) { ... }`

- Return parameters:

- `public isolated writable T M() { ... }`

- The hidden 'this' parameter for instance methods:

- `public isolated void M() writable { ... }`

- (The same locations for isolated delegates.)

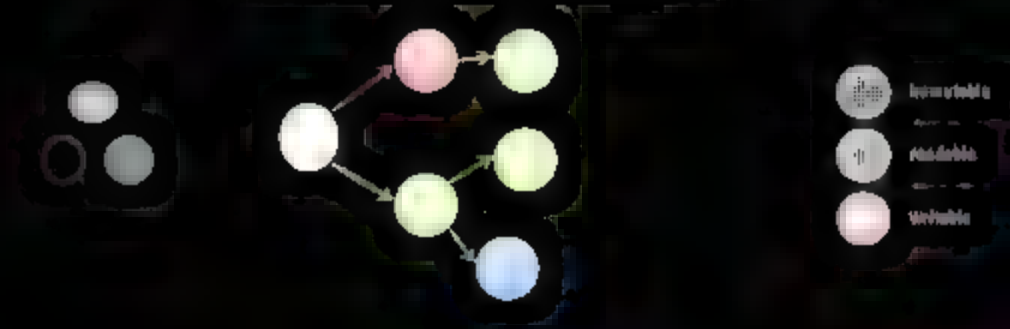
- Actual parameters for generics:

- `List<writable T> list = new List<writable T>();`

Permission combining

- Obtaining a field value combines permissions:
 - Accessing a field yields the least permissive of the field and the source reference.
 - E.g., writable U via a readable T \Rightarrow readable U.
- Ensures “deepness”, So:
 - Immutable Foo f: no mutable state accessible via f.
- Generics similarly combine between generic and instantiation.
 - E.g., writable T will yield least permissive between writable and concrete instantiation.

An Object Graph

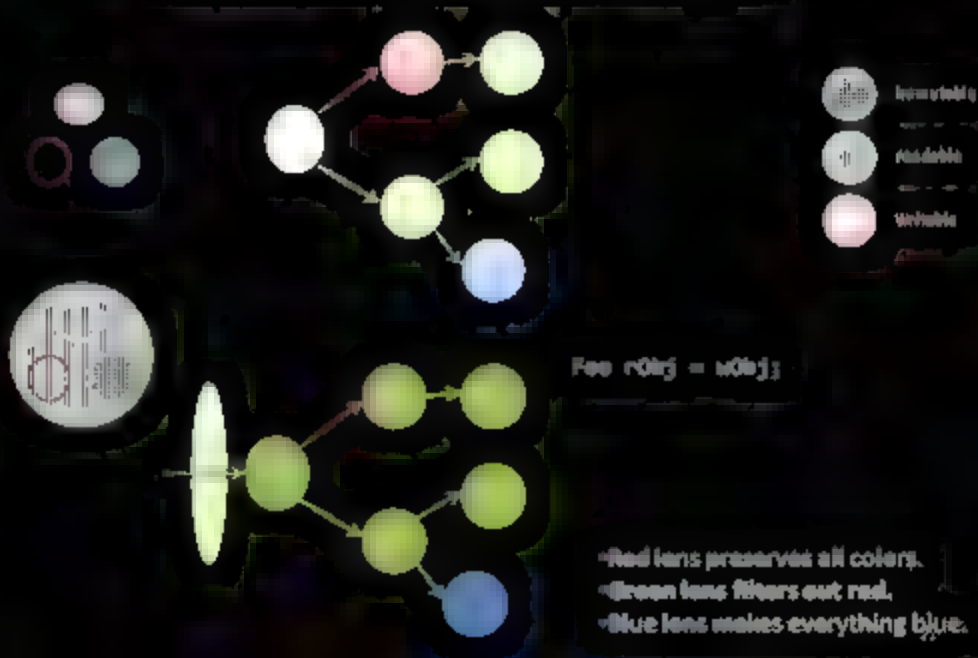


- Red lens preserves all colors.
- Green lens filters out red.
- Blue lens makes everything blue.

Writable Object Graph



Readable Object Graph



Immutable Object Graph



But Wait! There's More!

- Co- and contravariant permissions on override.

```
public abstract class A { abstract Immutable T f() writable; }  
public class B : A { override /*readable*/ T f() Immutable; }
```

- Local variable permission inference.

```
writable Immutable<T> o = d.GetImmutable();  
var o = d.GetImmutable();
```

- Anonymous method inference.

```
public delegate void D(T o);  
public isolated delegate void Isd(T o);
```

```
void M(D d);  
void M(Isd d);  
void f() {}  
isolated void g() {}
```

```
M(() => f()); // OK! Binds to M(D);  
M(() => g()); // OK! Binds to M(Isd);
```

Example: String Concatenation

No new annotations:

```
public string Concat(string[] ss) {  
    var sb = new StringBuilder();  
    foreach (string s in ss)  
        sb.Append(s);  
    return s.ToString();  
}
```

Relies on defaults; expanded code is:

```
public immutable string Concat(readable string[immutable] ss) {  
    writable StringBuilder sb = new StringBuilder();  
    foreach (immutable string s in ss)  
        sb.Append(s);  
    return s.ToString();  
}
```


Example: ICollection<T>

```
interface ICollection<T> {  
    bool Contains();  
    int IndexOf(T item);  
    void CopyTo(writable T[] array, int arrayIndex);  
    int Count { get; }  
    bool IsReadOnly { get; }  
  
    void Add(T item) writable;  
    void Clear() writable;  
    bool Remove(T item) writable;  
}
```

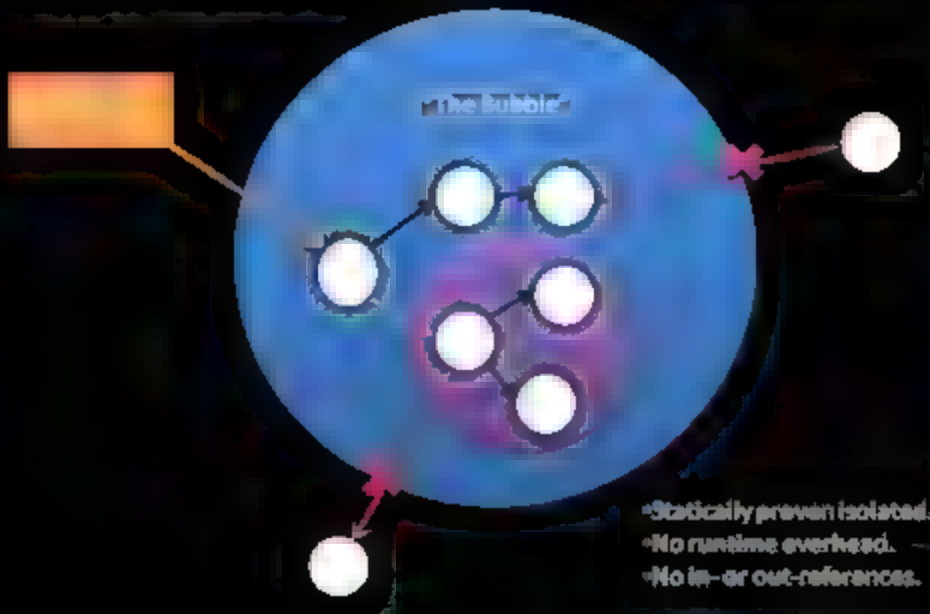
Isolation in Midori

- Everything above the line uses TSE by default.
 - No freethreading, memory models.
 - Small # of abstractions that don't.
 - E.g., Lazy<T>, data parallel, etc.
 - Below the line is opt-in.
- Single threaded processes.
 - Communicate via message passing.
 - Internal data parallelism.
 - ...But only the safe kind.



Isolated Object Graphs

```
delegate T PureFunc<T>() immutable;
```



Safe Parallelism

- Concurrency ground rules:

- No two threads ever see the same object as writable at once.
- No thread sees an object as readable while another sees it as writable.
- An object can always be seen as immutable.

- Two categories:



- Structured is split into two subcategories:

- In-place updates: e.g., arrays, isolated object graphs.
- Functional: e.g., LINQ-like data comprehensions.

A Few Examples

- Parallel Fibonacci:

```
public static int Fib(int n) {  
    if (n <= 1) return 1;  
    var result = Parallel.Invoke(() => Fib(n - 2), () => Fib(n - 1));  
    return result.Item1 + result.Item2;  
}
```

- LINQ-like comprehensions:

```
public static int[] Inverse odds(int[] input) {  
    return input.AsParallel().  
        Filter(x => (x & 1) == 1).  
        Map(x => -x).  
        Execute();  
}
```

any new conditions
are flagged as
combinable (and are not)

A Few Examples

- Parallel Fibonacci:

```
public static int Fib(int n) {  
    if (n <= 1) return 1;  
    var result = Parallel.Invoke(() => Fib(n-1)), () => Fib(n-2));  
    return result.Item1 + result.Item2;  
}
```

- LINQ-like comprehensions:

```
public static int[] Inverse odds(int[] input) {  
    return input.AsParallel().  
        Filter(x => (x & 1) == 1).  
        Map(x => ~input[x]).  
        Execute();  
}
```

any new conditions
are flagged as
completing this execution

over 9999% of the time only the variable being tested in the filter requires to be volatile

Benevolent Side Effects

"[T]here is no one right way of writing every program and a language designer has no business of trying to force programmers to use a particular style. The language designer does, on the other hand, have an obligation to encourage and support a variety of styles and practices that have proven effective and to provide language features and tools to help programmers avoid the well known traps and pitfalls."

*— Bjorne Stroustrup,
A History of C++: 1979—1991*

Benevolent Side Effects

System abstractions must lie sometimes:

- No mutable statics? `public static LazyFunc<T> m_func = new LazyFunc<T>();`

So long as they dynamically preserve the safety, `unstrict` can be used:

```
public class Lazy<T> where T : class {  
    private T m_value;  
    private LazyFunc<T> m_func = ...;  
    private object m_lock = new object();  
  
    public T Value {  
        get {  
            if (m_value == null)  
                unstrict {  
                    lock (m_lock)  
                        if (m_value == null)  
                            m_value = m_func();  
                }  
            return m_value;  
        }  
    }  
}
```


Talk Overview

- What is Midori?
- What is Mostly Functional C#?
 - Object Isolation
 - Other Features
- The Road Ahead: Reality Check
- Q & A

Immutable Types

- Marking any type as immutable:
 - All fields are readonly (unless marked mutable).
 - “Shallow” immutability.
- Property initializers:

```
immutable class InnStack<T> {  
    private static InnStack<T> Empty = new InnStack<T>();  
    public InnStack<T> Head { get; } = Empty;  
    public T Value { get; }  
}
```

- Deep can be obtained with immutable refs.

Non-Nullability

- “I call it my billion-dollar mistake. It was the invention of the null reference in 1965.”
 - Tony Hoare,
Historically bad ideas, QCon London 2009
- Only T? references may be assigned null.
 - Unification of nullable<T> for ref and val types.
 - T? coercable to T, but entails a dynamic null check.
- A lot like Spec# non-nullable types, T!, but the default is flipped.

Linear Types

- A reference can be marked linear:
 - `linear<Graph<T>> graph = _;`
 - No aliases allowed ~ exclusive ownership.
- Type must be marked linearizable:
 - No aliasing of 'this' during execution.
 - All fields are themselves linear.
- Allows:
 - Handoff of writable linear objects:
 - `writable linear<T> v = _;`
 - `Task<int> t = new Task<int>(o => o.Mutate(), v);`
 - Update in-place of linear objects:
 - `Writable linear<T> vs[writable] = _;`
 - `Parallel.ForEach(vs, v => v.Mutate());`

Specialization / Matching

- Static callsite specialization:
 - Pattern matching.
 - More efficient code generation.

```
public double Sum<T>(T x)
{
    where T : int, double {
        return x;
    }
    else where T : List<U> {
        return Sum(x.Car) + Sum(x.Cdr);
    }
    else {
        return 1.0;
    }
}
```

Async “Future” Lifting

- We use “promises” in Midori for async.
 - Any value can be a promise for a value.
 - Pipeline operations against it; or, invoke When and, in the callback, you can access the value.
- Any T can be marked `async<T>`:
 - `async<T> a <- Remote.Call();`
 - `async<U> b <- a.Foo(); // Pipeline.`
 - `async<V> c <- b.When(v => v ...); // When.`
- Notice that `<-` is used as “message send”.

Tuples

- Syntactic sugar for tuples (in .NET 4.0).
 - Usable as “anonymous” types that appear publicly.
- A first class type:
 - `(int, DateTime) ~ Tuple<int, DateTime>`
 - `public void M((int, DateTime) v) { .. V.First ...};`
- And special initializer syntax:
 - `var t = new () { 42, “hello”, 0.1f};`
 - `t` is of type `(int, string, float) ~ Tuple<int, string, float>`

Addressing Legacy (and Reality)

- **Bifurcation:**

- nonstrict methods don't abide by the rules.
- strict methods do.
- And only call other strict methods.

- **Downward separation:**

- Legacy (nonstrict) can call into new code (strict).
- Once you're in, you can't get out.

- **"We don't make things worse".**

- Pass an object as writable to concurrent strict methods.
- You had a race before. No worse.
- But we give you a pocket of safety: no mistakes.

Current Status

- Initial phase ("is it too crazy to work?"):
 - Integration with Axum's type system (via Q).
 - Custom attributes + FxCop verification done.
 - `[Writable] void Foo([Writable] Bar b) { ... }`
 - Prototypes of:
 - Midori base class libraries.
 - Safe data parallel programming model.
- Next phase ("get real"):
 - C# 3.5 compiler work 90% done.
 - Aim to wrap up in a few weeks; after that: all of Midori uses it.
 - Will go up on <http://www.microsoft.com/midori/>.
 - Data parallelism, race free finalization, message passing.
 - ...implicit in several areas (e.g., Linq, Array.Sort).

The Road Ahead: Challenges

- Is opt-in to imperative mutation cumbersome?
 - **writable** by default makes it easier.
 - But relies on abstinence to go functional.
 - *The bet*: mostly functional is the sweet spot; hence less ceremony.
- Safe parallelism is sometimes limiting:
 - No ad-hoc inter-thread communication.
 - *The bet*: race freedom is well worth it.
- What to do with the existing libraries...
 - Slowly move them on to the new plan?
 - But what about interfaces, events, callbacks, ...
 - And other languages who don't care: Python, Ruby, ...

Talk Overview

- What is Midori?
- What is Mostly Functional C#?
 - Object Isolation
 - Other Features
- The Road Ahead: Reality Check
- **Q & A**

Q & A

- <http://toolbox.mfcsharp/>
- <http://midori/>

Specialization / Matching

- Static callsite specialization:
 - Pattern matching.
 - More efficient code generation.

– E.g.,

```
• public double Sum<T>(T x)
{
    where T : int, double {
        return x;
    }
    ...
    else where T : List<U> {
        return Sum(x.Car) + Sum(x.Cdr);
    }
    else {
        return 1.0;
    }
}
```